Suppose we think of 4-5 algorithms & we've argued they always yield the correct answer.

How can we determine which will be most efficient, especially as \( n \) gets large?

Why not implement all of them & run them?
Problems
- Takes a lot of time to implement & test
- Time complexity often depends on input itself
- What data size should you use?
What does time complexity depend on?

- input itself
- input size ($n = 100$ vs $n = 1,000,000$)
- hardware (computer) \{ machine dependent \}
- compiler \{ machine dependent \}

Focus on a machine-independent analysis.
Asymptotic Time Complexity

machine independent, rough measure of time complexity as a function of input size \((n)\)

"Back of the envelope" calculation
What are we going to measure?

# of statements  (lines of code) executed

need to be careful that all statements in your high-level language take "roughly" the same time.
Input size vs Execution time
Input size vs lines of code

lines of code executed: merge sort vs. insertion sort

number of elements in array

Input size vs lines of code
How do we account for dependencies in the input?

Worst-case analysis — consider the input of the given size that is slowest

Expected-case (average-case) analysis — assume some distribution over the data
$n^2$ vs $n \log n$
Asymptotic Growth Rates

- $C$ constant
- $C \cdot \log n$ logarithmic
- $C \cdot n$ linear
- $n \cdot \log n$
- $C \cdot n^2$ quadratic
- $C \cdot n^3$ cubic

we'll give an algorithm for closest pair w/ this complexity